Cryptographic Verification of Test Coverage Claims

P. T. Devanbu
Dept of Computer Science,
University of California,
Davis, CA 95616 USA
+1-530-752-7324
devanbu@cs.ucdavis.edu

S. G. Stubblebine
Certco—Research
55 Broad St. - Suite 22,
New York, NY 10004, USA.
stu@cs.columbia.edu

February 26, 1999

Abstract

The market for software components is growing, driven on the "demand side" by the need for rapid deployment of highly functional products, and on the "supply side" by distributed object standards. As components and component vendors proliferate, there is naturally a growing concern about quality, and the effectiveness of testing processes. White-box testing, particularly the use of coverage criteria, is a widely used method for measuring the "thoroughness" of testing efforts. High levels of test coverage are used as indicators of good quality control procedures. Software vendors who can demonstrate high levels of test coverage have a credible claim to high quality. However, verifying such claims involves knowledge of the source code, test cases, build procedures, etc. In applications where reliability and quality are critical, it would be desirable to verify test coverage claims without forcing vendors to give up valuable technical secrets. In this paper, we explore cryptographic techniques that can be used to verify such claims. Our techniques have certain limitations, which we discuss in this paper. However, vendors who have done the hard work of developing high levels of test coverage can use these techniques (for a modest additional cost) to provide credible evidence of high coverage, while simultaneously reducing disclosure of intellectual property.

Form SF298 Citation Data

Report Date *("DD MON YYYY")* 26021999	**Report Type** N/A	**Dates Covered (from... to)** *("DD MON YYYY")*

Title and Subtitle Cryptographic Verification of Test Coverage Claims	**Contract or Grant Number**
	Program Element Number
Authors	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) IATAC Information Assurance Technology Analysis Center 3190 Fairview Park Drive Falls Church VA 22042	**Performing Organization Number(s)**
Sponsoring/Monitoring Agency Name(s) and Address(es) Defense Technical Information Center DTIC-IA 8725 John J. Kingman Rd, Suite 944 Ft. Belvoir, VA 22060	**Monitoring Agency Acronym**
	Monitoring Agency Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes	
Abstract	
Subject Terms	
Document Classification unclassified	**Classification of SF298** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** unlimited
Number of Pages 33	

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	2/26/99	White Paper

4. TITLE AND SUBTITLE
Cryptographic Verification of Test Coverage Claims

5. FUNDING NUMBERS

6. AUTHOR(S)
P. Devanbu and S. G. Stubblebine

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

IATAC
Information Assurance Technology Analysis Center
3190 Fairview Park Drive
Falls Church VA 22042

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Technical Information Center
DTIC-IA
8725 John J. Kingman Rd, Suite 944
Ft. Belvoir, VA 22060

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE
A

13. ABSTRACT ***(Maximum 200 Words)***

The market for software components is growing, driven on the \demand side" by the need for rapid deployment of highly functional products, and on the \supply side" by distributed object standards. As components and component vendors proliferate, there is naturally a growing concern about quality, and the e®ectiveness of testing processes. White-box testing, particularly the use of coverage criteria, is a widely used method for measuring the \thoroughness" of testing e®orts. High levels of test coverage are used as indicators of good quality control procedures. Software vendors who can demonstrate high levels of test coverage have a credible claim to high quality. However, verifying such claims involves knowledge of the source code, test cases, build procedures, etc. In applications where reliability and quality are critical, it would be desirable to verify test coverage claims without forcing vendors to give up valuable technical secrets. In this paper, we explore cryptographic techniques that can be used to verify such claims. Our techniques have certain limitations, which we discuss in this paper. However, vendors who have done the hard work of developing high levels of test coverage can use these techniques (for a modest additional cost

14. SUBJECT TERMS
Software, Cryptography, Encryption, Information Security

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	UNCLASSIFIED	UNCLASSIFIED	None

1 Introduction

As the size, functionality, and complexity of software applications increase (*e.g.* the Microsoft Office^{TM} products are in the range of 10^6 lines) vendors seek to break applications into components (spell checkers, line breakers, grammar checkers etc.). Distributed component standards such as CORBA, ubiquitous networking, portable object-oriented platforms such as Java are also additional drivers of this trend towards software components. A vibrant market for software components is growing. The cost of entry into this market is low, and small vendors can be players. The prospect of achieving lower costs *and* higher quality with commercial off-the-self (COTS) components has wide attraction. However, as the number and types of components proliferate, and smaller, newer vendors enter the market, there is a natural concern about quality.

Recently, there has been great interest in the use of COTS in safety-critical systems. A special section of *IEEE Computer* (See *IEEE Computer*, June 1998) has been dedicated to this issue. More specifically, the issue has been taken up in the transportation [26] and nuclear power [33] fields. The use of COTS components is driven by economic considerations. However, there is deep concern about the risks of using COTS components (that may have been developed for the consumer market) in safety-critical systems. Traditionally, the procurement processes used by customers of safety critical systems requires vendors to disclose details about the processes used to develop the system, as well as product-related information such as requirements, design documents, source code, test cases, etc. However, COTS vendors, who are facing highly competitive markets, risk loss of valuable intellectual property if they disclose such details. Recently a committee appointed by the National Research Council (NRC) in the USA (See [31], pp. 71-76) has discussed the reuse of COTS software in nuclear power plants. Their report states (Page 76, first para):

> "Dedication of commercial components requires much more information than commercial vendors are accustomed to supplying ... Some vendors may be unwilling to provide or share their proprietary information, particularly about development or testing procedures and results of service experience ..."

Speaking on the same issue, Voas ([36], page 53) states:

> "Software components are delivered in "black boxes" as executable objects whose licenses forbid de-compilation back to source code. Often source code can be licensed, but the cost makes doing so prohibitive."

Thus, potential users of COTS components find themselves in a quandary. On the one hand, due to the competitive market for COTS components, re-using them can lead to huge savings in cost and time-to-market. On the other hand, COTS vendors facing these tough markets cannot risk giving up their competitive advantage by disclosing information that COTS users *need* to verify that the components are of adequate quality. While this issue is particularly

acute for safety-critical systems, it is also relevant to other cases, for example to the use of COTS components in 7x24 business-critical electronic commerce servers. See Voas [36] for a more detailed and lucid examination of this topic[1].

Traditionally, systems with stringent quality requirements undergo a rigorous verification process, often under the auspices of third party *verification agents* [4, 23, 24]. One common testing technique used is *white-box* testing; The goal is to ensure that a system has been adequately exercised during testing. In this approach, an abstraction of the system (*e.g.*, a control flow graph) is used to identify the parts of the system that need to be exercised. Each of these parts is called a *coverage point*, and the entire set is the *coverage set*. When a suite of tests can exercise the entire coverage set, it is called a *covering test suite* with respect to this coverage set. A sample criterion for adequate test coverage is that most of the basic blocks in the control flow graph have been exercised. Typically, test coverage is verified by building an "instrumented" version of the system; the instrumentation is placed in the appropriate parts of the system; the added instrumentation makes a persistent record when coverage points are executed. Thus it is possible to verify that the necessary parts of the system have been exercised. We can abstract the situation as follows:

There is a system X, with source code S, out of which a (shipped) binary B_s is built. Also, from the source S, we can generate a coverage set C_γ (for some coverage criterion γ) as follows:

$$C_\gamma = \{c_1, c_2, \ldots c_n\}$$

Each of the c_i's refer to a coverage point. To cover these coverage points, it is necessary to develop a covering test suite T_γ that can exercise the coverage set for the criterion γ:

$$T_\gamma = \{t_1, t_2, \ldots t_m\}$$

such that for any given coverage point c (or perhaps for most coverage points c) there is a t such that the execution of test t hits c. This is verified by first building an appropriately instrumented binary B_γ, and running the test suite T_γ.

A vendor who undertakes the cost of developing an adequate set T_γ for some stringent γ can reasonably expect that the system is less likely to fail[2] in the field due to undetected faults in system X [20]. Often, in fields with exacting reliability requirements (such as transportation, telecommunications, energy or health) software users demand high quality standards, and expect vendors to use testing processes that achieve high levels of coverage with stringent coverage criteria. In such situations, in order to establish that the requirements have been achieved, a vendor may balk at giving a customer access to the source code and test scripts so that coverage can be verified.

[1] Voas proposes a solution to this quandary based on black-box testing; we use white-box testing. When we discuss related work, we examine the relation of this research to our work in more detail.

[2] Providing, of course, that the system passes the tests!

The goal of this paper is to propose solutions based on cryptographic techniques to allow software vendors to convince their customers that high levels of test coverage have been achieved, while protecting intellectual property.

2 Background and Related Work

In this section, we first review background concepts in white-box (coverage) testing. We then explore prior work on verifying the quality of COTS components. Most of this work has been based on black-box testing, while our work emphasizes white-box testing. We then describe the currently available approaches for white-box test coverage verification and their advantages and limitations. This leads to an analysis of the threat models that are applicable to test coverage verification protocols. We then present a brief background on the cryptographic techniques we use in this paper.

2.1 Coverage Testing

Coverage testing is a well-defined notion that has received a lot of attention in the literature. Several different criteria (γ in the discussion above) have been proposed; the relationships between these criteria have been well studied [16, 17]. Many are based purely on control flow, such as basic block coverage, branch coverage, and path coverage. There are several different criteria that are based on data flow–i.e., the definition and use of values stored in memory. The data flow testing criteria are difficult to use with programs that involve the use of arrays, pointers, and other types of aliasing. Since most large applications use heap-allocated memory addressed through pointers, it is non-trivial to apply data flow test coverage criteria. The precise definition of all of the testing criteria can be found in the literature, as well as various models for comparing their relative strengths [17, 16]. Coverage testing is very popular in industry; this has given rise to a rich and competitive market for tools that support coverage testing.

Another approach to testing is statistical or random testing [12], whereby a very large sample of tests drawn from a "realistic" distribution is applied to the system. Defects found in the testing process are iteratively fixed as the process proceeds. The goal is to get the system to run over very large input samples without errors. Finally, an estimate the relability of the system subject to statistical confidence intervals can be computed. Typically, 10^6 or 10^7 tests are required to achieve the reliability levels that are desired in markets with a need for high-quality software (such as transportation, telecommunications, energy or health). Verification of random testing processes is beyond the scope of this paper.

These approaches have been justified by empirical work. Experimental work [14, 20, 27] indicates that branch coverage levels in the range of 85-95% have a high likelihood of exposing remaining faults in the software after development. Branch and block coverage testings will

form the primary focus of our discussion. The extension of our techniques to data flow testing remains a subject of future work.

2.2 Black-box approaches to COTS verification

Voas [36] considers the impact of introducing a COTS component into an existing system. How can one evaluate the quality impact of this introduction? Voas assumes that it would be impossible to use white-box approaches due to unavailability of enough information about the system. He recommends 3 assessment techniques to evaluate the suitability of a candidate COTS component for a particular application (All 3 techniques treat the COTS component as a black box):

1. **Black box testing** The component *per se* is tested to see if it performs correctly using the embedding system's operational profile; i.e., does the component perform adequately based on the mostly likely use scenarios, given the embedding context. The COTS user creates a test driver, and a test oracle [37] (to decide if the output of a test is correct). This approach tests components as is, without knowledge of internals. However, there are several disadvantages; a very large number of tests may have to be run to get an accurate estimate of a component's reliability. If an automated oracle cannot be constructed, this can be expensive.

2. **System-level fault injection** This approach to COTS validation, considers the impact of a misbehaving COTS component on the entire system. Misbehaviour is introduced by perturbing the input/output relationship of the COTS component within the embedding system. If the embedding system tolerates such perturbations, then failures in the COTS component can be tolerated. This approach emphasizes the robustness of the overall system, and reduces the impact of low-quality COTS. However, the costs of increasing the robustness of the entire system must be traded-off carefully against the cost savings achieved by using COTS in the first place.

3. **Operational system testing** This is the standard operational-profile based testing, which uses large-scale system level testing to develop good estimates of overall system reliability in the presence of COTS components. This approach exercises the system with many random samples drawn from a distribution which matches "typical use". The disadvantage is that huge sample sizes are required to provide good estimates for highly reliable systems, and this can be expensive and time consuming.

Our approach (described below) is a white-box approach, based on coverage testing and complements the approaches suggested by Voas. We provide a way for vendors who have achieved high levels of test coverage to provide credible evidence of their achievement to customers, while protecting their intellectual property to a large degree.

2.3 Current Approaches to Coverage Verification

Currently, test coverage verification is done by a third party testing which is trusted by both the vendor and customer to operate according to well defined procedures. Several commercial laboratories [4, 23, 24] provide this service. We now describe how this works. First a little notation: in all the following scenarios, $\mathcal{V}$ refers to the vendor who claims to have achieved γ test coverage on a system X, and $\mathcal{C}$ refers to the skeptical customer who wants to be convinced and $\mathcal{T}$ refers to a third party trusted by both $\mathcal{V}$ and $\mathcal{C}$.

Basic Third Party Method

1. $\mathcal{V}$ sends $\mathcal{T}$, a trusted third party, the source code S, and the test suite T_γ, and a description of the coverage criterion γ.

2. $\mathcal{T}$ builds binary B_γ from S, and constructs the coverage set C_γ.

3. $\mathcal{T}$ runs the test suite T_γ against B_γ and verifies that the suite hits the coverage set.

4. $\mathcal{T}$ tells $\mathcal{C}$ that the γ coverage criterion for X has been met by the test suite T_γ.

5. $\mathcal{V}$ ships B_s to $\mathcal{C}$, with the claim that $\mathcal{T}$ has verified coverage.

This approach has a number of limitations, including additional costs and delays of third party involvement, as well as the difficulty of finding an appropriate trusted third party. These are explored further below (Section 2.4.6). In the sequel, we also refer to the above method as the *basic method*.

However, in general, when absolute trust is required, a reliable third party is available, the vendor is willing to risk full disclosure to this party, and cost and delay are not factors, this is a workable approach, and indeed, is in current use. Our goal is to explore various techniques that can be used to reduce disclosure, lower costs, reduce delays, and in some cases eliminate the third party. Before we do this, we will adopt a general perspective on the problem of test coverage verification, and discuss threat models (possible ways in which parties to the test coverage verification process may behave in order to gain an undesirable advantage). While we cannot address all the different threats we discuss, our work does counter many of them.

2.4 Threat Analysis

We now analyze the various possible ways in which test coverage verification protocols can be compromised. We first describe our method for analyzing threats to test coverage verification, and then we list a set of threats that we have considered.

2.4.1 Caveats and Assumptions

This work rests on several assumptions.

First, we assume that vendors are strongly motivated by market forces[3] to provide the highest quality software. Our goal is to *allow vendors to face this market demand for high quality software by providing customers with credible evidence of high quality software practices, while reducing disclosure of intellectual property.*

Second, coverage testing (like any other testing method) can be used in different ways. We address block testing or branch testing; other approaches will be dealt with in future work. In addition, no form of coverage testing is perfect. Weaknesses of coverage testing can lead to vulnerabilities in test coverage verification protocols. These issues are dealt with greater detail in Section 2.4 and 5, and some strategies for dealing with these problems are suggested; however, a significant practical disincentive to such practices is the market demand for high quality software, and the high cost of cancelled sales, refunds and/or fixing bugs in the field. Third, we make the usual assumption that all parties involved in coverage verification protocols have access to a test oracle[37] that decides whether the output for any test case is right or not. In our case, this assumption is a fairly mild one. Our verification protocols typically require only a limited number of test cases to be run, compared to total number of test cases required to achieve white box test coverage. Given this limited number, even a "manual" oracle is practical.

Finally, we have little to say about techniques for defeating reverse engineering; our focus is more to protect the secrecy of the largest possible number of test cases (which represent a significant investment by the vendor), while allowing the use of common methods for repelling reverse engineering, such as shipping only binaries without symbol tables. Without a physically protected hardware platform, a determined adversary can reverse-engineer a good deal of information about software. Techniques and tools to support reverse engineering are an area of active research. In fact, previous research [7, 9, 25, 32] demonstrates how control-flow graphs, profile information, compiler-generated binary idioms, and even slices can be derived by analyzing and instrumenting binaries. De-Compilation (converting binary to source code) and binary porting (converting binaries from one machine architecture to another) are typical goals of binary analysis. We say more on this in the conclusion.

2.4.2 Methodology

We now describe our methodology for analyzing threats to our protocols. The focus of security here is not at the session layer but at the application layer. Thus we assume the testing protocols occur over secure channels[4] such as those provided by Secure IP (IPSEC

[3] Different kinds of software sell in different markets, so quality needs do differ. For example, software components used in nuclear power plants need to be of much higher quality than those used in desktop word processors.

[4] Such facilities provide message integrity, confidentiality, and data-origin authentication.

[6]) and the Secure Socket Layer (SSL [18]) protocols. We do not consider weaknesses in these protocols.

Traditionally, the analysis of security protocols at the application level follows a couple of basic paths. A basic approach is to specify the high level protection objectives as state invariants and to specify all possible actions of the adversary (i.e., state transition rules). One would go on to show that no possible sequence of actions can lead to a violation of the security property. However, to adopt this approach, one needs a *provably complete* characterization of all possible actions of the adversary. In a new application domain such as test coverage, it is impossible to know if one has considered a complete set of attacks.

However, we introduce several novel types of threats that arise in the domain of test coverage verification, and describe our defenses against these threats. There may be others, and in general it is impossile to know what they might be. However, our approach of consideration of a set of threats is typical in the security community (see, for example [3, 30]; there are numerous other examples). Protocol specifications are refined to resist attack scenarios known at development time, and then they are published or deployed. Subsequently, other attacks may be discovered, which may lead to further refinement of the protocol. Our approach follows this tradition.

2.4.3 Attack Modes

Test coverage verification involves several steps, each of which can be attacked.

1. The shippable binary B_s, and an instrumented binary B_γ are built from the source S, using appropriate compilers and other tools
2. The coverage set C_γ is derived from S.
3. The vendor, using manual or other means, creates the covering test set T_γ.
4. The test set T_γ, or some randomly selected portion thereof, is run against B_γ to ensure coverage—this is done by some acceptable party.
5. The binary B_s is shipped.

All these steps must work correctly in order to verify test coverage correctly, as should the tools that are used. Note that in the above important items are disclosed: the source code (used to build the coverage test and the binary), and the test cases. Our goal is to protect as much of this information as possible. However, approaches to protect this information must resist attempts by $\mathcal{V}$ to boost the coverage level through deception or by $\mathcal{C}$ to gain proprietary information.

We now discuss several attacks, risks and difficulties with test coverage verification. These can be roughly characterized into three categories: *process related* (**P.1, P.2, P.3**), *sampling*

related (**S.1, S.2**) and *testing related* (**T.1, T.2**). The attacks *P.1.a, P.1.b* involve attempts by $\mathcal{V}$ to incorrectly operate the test coverage verification process in order to trick $\mathcal{C}$. **P2** and **P3** discuss risks to the $\mathcal{V}$ who undertakes test coverage verification. The sampling related attack **S.1** exploits statistical variation in random samples; **S.2** uses artificial (non-random) samples to force $\mathcal{V}$ to disclose information. We suggest some simple, practical techniques to deter these two types of attacks. We have built some tools for use in conjunction with the protocols we describe. The testing related attacks **T.1, T2** try to exploit inherent weaknesses in coverage testing. We offer some techniques to deter these attacks; however, there are some complications; we're continuing to investigate better approaches.

2.4.4 P.1 Bait and Switch Attacks

Issues **P.1 & P.2** both have to do with protecting information. **P.2** is concerned with risks to the $\mathcal{V}$ if s/he discloses too much information; **P.1** is concerned the risks to $\mathcal{C}$—*i.e.,* ways in which $\mathcal{V}$ might cheat if s/he were not required to disclose all the relevant information.

One attack (by $\mathcal{V}$) (*P.1.a*) would be to simply switch the shipped binary. The vendor might contrive to thwart the process by submitting a phony version of the binary (which is perhaps far simpler and smaller, and easier to test) for coverage testing, and ship another version to the customer. Thus, in the approach described above (that is currently in use) the vendor could try to ship $\mathcal{T}$ a smaller, phony version of the binary, and a correspondingly smaller (and easier to create) set of tests, and try to gain high coverage ratings. He could then ship the real (much larger) binary claiming the high coverage results obtained from the phony binary. This attack may also include the shipment of a phony source code to go with the phony binary.

However, vendors are typically reluctant to reveal the source code. In order to protect the source code, we develop approaches where coverage points are generated by the vendor, from the source code, and sent to the third party or the customer . In such cases, another version (*P.1.b*) of this "bait and switch" attack could be used. Some of the coverage points, specially ones that are harder to reach, could be removed from the delivered coverage set. This will make the coverage measure seem larger than it really is. Even if the vendor is forced to use a tool that actually does correct coverage analysis, he could tamper with the tool and compromise it.

2.4.5 P.2 Disclosure Hazards

Much of the work described in this paper is aimed at reducing the amount of information that the vendor has to disclose, such as source code, symbol tables, and (particularly) the test cases. Source code is clearly the most valuable information; the symbol table, which is typically embedded in the binary, can also be a valuable aid to reverse engineering. In some cases, the set of coverage points might itself reveal valuable information about the

structure and complexity of the program, and perhaps even the amount of structure devoted to a specific function. A large and exhaustive set of test scripts is also valuable information regardless of whether the source code or symbol table is known. Generating such a test set involves careful analysis of the requirements, as well as familiarity with the design and implementation of the system. While the most typical functions of the system may be widely known, a complete set of test cases would have to exercise unusual situations, create various feature interactions, cause exceptions to be raised, etc. A comprehensive consideration of all these special cases is a valuable piece of intellectual property that demands protection. Indeed, there are vendors who make it their business to develop and sell comprehensive test suites [21, 35]. Our goal is to protect this information.

2.4.6 P.3 Who does the verification?

The conductor of the coverage verification process needs to be trusted. That is, the tests claimed to achieve coverage have to be run, and the system has to work correctly; the actual coverage level achieved has also to be measured. This entire process has to be conducted in a manner that inspires trust in the customers of the software. First, there may not be a trusted third party acceptable to $\mathcal{C}$, and to whom $\mathcal{V}$ is willing to to reveal the source and the test suite. There may be a commercial incentive for $\mathcal{T}$ to exploit this information in some legal, unexpected or undetectable ways; a truly trustworthy $\mathcal{T}$ may be hard to find. Second, $\mathcal{T}$ has to replicate $\mathcal{V}$'s entire build apparatus, (*i.e.,* compilers, libraries, and auxiliary tools) and execute the build (which may be complex). Third, $\mathcal{T}$ has to be at least as aware of application area as $\mathcal{C}$ — enough to judge if the response of the system to a given test is acceptable. Finally, the approach will lead to additional costs, delays, etc., due to the need for an additional step by another party prior to customer acceptance.

2.4.7 S.1 Prospecting

In some of the protocols we describe below, we seek to protect the largest possible number of coverage points and test cases. We advocate protocols using random challenges, and estimate the actual coverage using statistical approaches. With any process based on random sampling, there is a confidence interval that represents the error range in the estimate. Thus if the vendor does the sampling, and the sampling process is not carefully controlled and monitored, a "prospecting" attack would be applicable: the adversary can simply keep repeating the sampling process until a biased sample is obtained, which tilts the estimate in the desired direction.

2.4.8 S.2 Targeted Sampling

On the other hand, if the sampling is in the hands of an adversary of the vendor, sampling could be conducted in a manner that is designed to force the vendor to disclose information.

For example, if an adversary desires to investigate the details some particular subsystem, or the implementation of some particular functionality, the "sampling" regimen could artificially target its focus on the area of interest, thus forcing the vendor to reveal information concerning a specific area of functionality or architecture.

2.4.9 T.1 Fault Hiding

Coverage testing is not perfect. A test case may exercise a coverage point (e.g., a basic block) that contains a fault, but it may not expose the resulting failure. For example a basic block may contain an unguarded computation of a square root. A failure would only result with a negative input; a positive input may cover that block, without exciting a failure. This weakness of coverage testing leaves vendors with the option of carefully selecting covering test cases that *hide* faults. As we shall see below, this attack causes the vendor the additional financial risk of deploying code that is likely to incur considerable maintenance costs. There are also some approaches to discouraging this attack.

2.4.10 T.2 Padding

The vendor may contrive to artificially boost coverage by adding spurious coverage points to the program that are very easily covered. For example, if the vendor has 1000 coverage points in the program, and has covered 700 of them, he could add 500 dummy coverage points to the program that are readily covered, and thus boost his perceived coverage artificially from 70% to 80%. Again, this approach has results in increased economic risks for the vendor—below 80% coverage, chances of faults slipping through are much higher. Fixing faults after releasing software is very expensive, and can damage the vendor's reputation; if a fault is found, there is a strong economic incentive to fix it as soon as possible. We also discuss some additional approaches to discourage this attack.

2.5 Brief Summary of Cryptographic and Interactive Techniques

Different combinations of customers, vendors and software require different solutions for the test coverage verification problem. In this paper, we apply several cryptographic techniques to address some common scenarios that may arise in practice. We now briefly describe the techniques we have used in our work; more complete descriptions can be found in [29]. All of these techniques are used to assure customers ($\mathcal{C}$) that the vendor($\mathcal{V}$) has high test coverage, while attempting to protect $\mathcal{V}$'s secrets. Our work is related to the notion of zero-knowledge protocols (ZKP). Zero-knowledge protocols are designed to allow a prover to demonstrate knowledge of a secret while revealing no information about the secret. ZKP are often based on a cut-and-choose challenge-response regime. We use a similar idea: a skeptical $\mathcal{C}$ challenges $\mathcal{V}$ to provide test cases, given a particular part of a system; with appropriate responses, $\mathcal{C}$ develops confidence in $\mathcal{V}$'s claims about test coverage. Unlike ZKP's however, our approach

actually reveals *some* information. The thrust of this work has been to reduce the amount of information revealed.

For the descriptions that follow, we use some public/private key pairs: assume $K_{\mathcal{V}}^{-1}$ is a good private signing key for the individual $\mathcal{V}$ and $K_{\mathcal{V}}$ is the corresponding public signature verification key for the lifetime of the test coverage verification process.

1. **Digital Signatures** Given a datum δ, $\sigma_{K_{\mathcal{V}}^{-1}}(\delta)$ is a value representing the signature of δ by $\mathcal{V}$, which can be verified using $K_{\mathcal{V}}$. Note that $\sigma_{K_{\mathcal{V}}^{-1}}(\delta)$ is not the same as encrypting δ with $K_{\mathcal{V}}$; typically, it is just an encrypted hash value of δ. In particular, δ cannot be reconstructed from $\sigma_{K_{\mathcal{V}}^{-1}}(\delta)$ with $K_{\mathcal{V}}$; however, given δ, the signature can be verified. This is a way for $\mathcal{V}$ to "commit" to a datum in an irrevocable manner.

2. **Signature Blinding** It is impossible to recover the value of a δ given the value of a signature $\sigma_{K_{\mathcal{V}}^{-1}}(\delta)$. However, given that two signatures are equal, one can guess that they both result from signing the same data item. To deny an adversary this information, it is possible to commit to a data item δ by generating a random number ρ and revealing the signature $\sigma_{K_{\mathcal{V}}^{-1}}(\delta \mid\mid \rho)$. When the time comes to reveal the input, both δ and the random number ρ are revealed. This denies the adversary the ability to infer equivalence classes among a set of commitments; however, the signatures remain trustworthy, irrevocable commitments.

3. **Trusted Third Party**. If there is a trusted third party (denoted by $\mathcal{T}$) that can act as a "buffer" between $\mathcal{C}$ and $\mathcal{V}$, $\mathcal{T}$ can use information supplied by $\mathcal{V}$ to assure $\mathcal{C}$ about $\mathcal{V}$'s testing practices, while protecting $\mathcal{V}$'s secrets. $\mathcal{T}$ can be relied upon both to protect $\mathcal{V}$'s secrets, and operate fairly, without bias. Note that $\mathcal{T}$ can certify a datum δ by appending the signature $\sigma_{K_{\mathcal{T}}^{-1}}(\delta)$.

4. **Trusted Tools**. To verify any type of white-box coverage, it is necessary to provide information extracted from source files. Tools (trusted by all parties involved) that extract exactly the information required for coverage verification, and adjoin a cryptographic signature with a published key, can be used to extract trusted coverage information, providing the tools are not tampered with.

5. **Random Sampling**. Consider an adversarial situation where a party $\mathcal{V}$ claims that a proportion ρ of the members of a set σ have a property π, and $\mathcal{V}$ is willing to reveal the proof that π holds for only *some* members of σ. A skeptical party $\mathcal{C}$ can choose a member σ_i, of this set at random, and challenge $\mathcal{V}$ to demonstrate that $\pi(\sigma_i)$ holds. After several such trials, $\mathcal{V}$ can estimate ρ subject to confidence intervals. This technique can be used to estimate test coverage levels.

6. **Autonomous Pseudo-Random Sampling**. The random challenges discussed above can be performed by $\mathcal{V}$ herself, using a published pseudo-random number generator $\mathcal{G}$

with a controlled seed. If $\mathcal{G}$ can be trusted by adversaries to produce a fair random sample, $\mathcal{V}$ can publish a credible, self-verified estimate of ρ.

2.6 The Solution Space

Our purpose in this paper is to explore a space of solutions to the problem of test coverage verification. While considering a specific case of test coverage verification, the following questions are important to address:

1. How much information (source code, coverage set, test scripts etc.) is the vendor willing to reveal?
2. What is the type of coverage that has to be verified?
3. Is there a third party trusted by both the vendor and the customer?
4. How much time/money can be spent on test coverage verification?
5. To what level of confidence does the customer want to verify test coverage?

These questions represent different goals, which are sometimes conflicting; any practical situation will surely involve trade-offs. Different trade-offs will be acceptable under different circumstances. This paper describes one set of possible solutions that cover some typical cases; we certainly have not covered all the possibilities.

3 Defending against process-related threats

The third-party protocol described above (Section 2.3, page 6). suffers from both both high *disclosure* and high *cost* (**P2, P.3**). $\mathcal{V}$ has to disclose the source code, the build apparatus, and the test cases. $\mathcal{T}$ then has to build an instrumented version of the binary, run the test cases, ensure the test cases work correctly, and measure the coverage; all this effort will add to $\mathcal{V}$'s costs. Reducing the disclosure from $\mathcal{V}$ increases the process-related threats *P.1.a,P.1.b*; the less information is disclosed, the more opportunities $\mathcal{V}$ has to cheat.

3.1 Verifying coverage without Source Code

We now explore some approaches that use just the binary to verify test coverage claims, and do not require $\mathcal{V}$ to disclose the source. We begin by assuming binaries built with full symbol tables, i.e., compiled with the "`-g`" option in most `C` compilers; the issue of eliminating (or abridging to the extent possible) this symbol table is visited later in this section.

Protocol 1

1. From S, $\mathcal{V}$ constructs the system B_s and the set of coverage points C_γ.
2. $\mathcal{V}$ sends $\mathcal{T}$: B_s, C_γ and the test suite T_γ, with the locations in the source files corresponding the coverage points.
3. $\mathcal{T}$ uses a binary instrumentation tool, either interactive (*e.g.,* a debugger, or batch-oriented (*e.g.,* ATOM [34], EEL [25]) to instrument B_s, using the line number/file information sent by $\mathcal{T}$, and the symbol table information embedded in B_s. For example, a debugger, can set break points at the appropriate locations (e.g., line numbers in files).
4. $\mathcal{T}$ runs the test suite T_γ against the instrumented binary, and verifies coverage level. For example, the debugger can be set to delete each break point when "hit". The coverage level is verified by the number of remaining breakpoints.
5. $\mathcal{T}$ signs the file B_s (perhaps after extracting the symbol table from B_s) and sends $\sigma_{K_{\mathcal{T}}^{-1}}(B_s)$ to $\mathcal{V}$.
6. $\mathcal{V}$ verifies the signature on $\sigma_{K_{\mathcal{T}}^{-1}}(B_s)$, using $K_{\mathcal{T}}$; then, $\mathcal{V}$ sends B_s and $\sigma_{K_{\mathcal{T}}}(B_s)$ to $\mathcal{C}$.

This method improves upon the *Basic Method* in a few ways: first, the source is not revealed to $\mathcal{T}$. Second, $\mathcal{T}$ does not recreate $\mathcal{V}$'s build environment. Third, $\mathcal{T}$ works with the shipped version of the software, so he can directly "sign" it after he has verified the coverage claims. This avoids the *P.1.a* (Section 2.4.4) type of attack; $\mathcal{V}$ cannot reproduce this required signature on a phony binary. Finally, $\mathcal{T}$'s work is reduced; rather than building instrumented and uninstrumented versions, he only has to instrument the binary, which is not harder than the link phase of a build. So presumably, *Protocol* 1 would be cheaper, faster and less error-prone than the *Basic Method.* A major weakness in *Protocol* 1 is that $\mathcal{V}$ is trusted to build an accurate coverage set C_γ. In other words, attack *P.1.b* (Section 2.4.4) is applicable. However, $\mathcal{V}$ risks alienating customers by shipping faulty software while falsely claiming high levels of test coverage. If the software fails frequently in the field, he could be called upon to reveal the source code to a trusted third party, and prove that his coverage analysis was accurate, and that the shipped binary was built with the same source code. This may dissuade large vendors (with much to lose), but not smaller ones. Now we describe an approach where trustworthy coverage sets could be generated by untrusted parties.

In the following protocol, $\mathcal{V}$ uses a trusted coverage analysis tool to generate the coverage set C_γ which is cryptographically "signed" by the tool. Now, $\mathcal{T}$ can verify the signature on the coverage set, and be assured that $\mathcal{V}$ has not generated a spurious (presumably smaller) coverage set.

Protocol 2

1. $\mathcal{V}$ uses a trusted coverage analysis tool, which generates the covering set from the source S. This tool analyzes the source code, to find the requisite coverage points for γ and generates a set of individually signed coverage points.

2. Proceed as *Protocol* 1, with $\mathcal{V}$ sending $\mathcal{T}$ his signatures of the coverage points, and $\mathcal{T}$ checking the signatures against the coverage points.

The main advantage of this approach is an increased level of trust in the set of coverage points generated by $\mathcal{T}$. The disadvantage is the risk that the trusted tool may be compromised. This issue is discussed later in the section.

As noted in the beginning of this section, the "binary-patching" approaches used in *Protocols* 1 and 2 both assume that the binary is shipped to $\mathcal{T}$ with the symbol table information intact. We now explore the implications of this, and approaches to eliminating the symbol table.

Binary instrumentation tools such as debuggers use the symbol table in the binary to locate machine instructions corresponding to source level entities. This information is used by debuggers to set breakpoints, to inspect the elements of a structure by name, etc. The symbol table has information about global symbols (functions/entry-points and variables), source files, and data structures (sizes, elements etc.). With this information, one can reverse-engineer a good deal of information about the implementation. Typically, symbol tables are stripped out of delivered software; as we discussed earlier, $\mathcal{T}$ could easily perform this stripping after verification. Sometimes, the $\mathcal{V}$ may balk at revealing this symbol table to $\mathcal{T}$. Can we verify test coverage without revealing the symbol table? *Protocol 3* addresses this issue. It uses a *pure binary patcher* such as `gdb`, which can insert break points at specific binary offsets. A command such as `break 0x89ABC` to `gdb` will set a break point at the machine address `0x89ABC` in the program. A batch-oriented tool like `EEL` [25] can also be used. Such a tool will be used by $\mathcal{T}$ to insert instrumentation at coverage points, and verify coverage. We also use *a binary location finder (blf)*, which uses the symbol table to find binary addresses for the coverage points. `gdb` can perform this function. For example, the "`info line` *file:line*" command in `gdb`, for a given a line number in a file, calculates the corresponding binary position and size. Such a tool will be used by $\mathcal{V}$ to identify coverage points by physical addresses in the binary; this tool would have to be trusted by $\mathcal{T}$, and would cryptographically "sign" its output.

Protocol 3

1. $\mathcal{V}$ uses a trusted coverage analysis tool (which signs its output) and generates the coverage set C_γ.
2. $\mathcal{V}$ then uses a binary location finder (blf) to find binary locations corresponding to each element of C_γ. Call this set $f_{blf}(C_\gamma)$. We assume that the blf signs its output; we can also have blf verify the signature of the coverage analysis tool from the previous step.
3. The protocol proceeds as before, in Protocol 2.

This approach reduces the amount of information revealed to $\mathcal{T}$ about the symbol table and the source file. However, it increases the reliance on "trusted tools." In addition to the

coverage analyzer used above, $\mathcal{T}$ needs to trust the binary location finder. This may not always be acceptable.

Trusted tools that "sign" their output are likely targets of attack due to commercial incentives to cheat. The coverage analysis tool could be modified and made to sign spurious, small coverage sets; a binary location finder could be made to always point to false locations; the "secret" key could be stolen from the tool and used to sign false data. One way to avoid this problem is through the use of a physically secure co-processor. Various forms of these processors are available in tamper-proof enclosures, running secure operating systems; they are expected to become ubiquitous in the form of *smart cards* [2, 38], which are expected to become quite powerful in a few years. The physical enclosure is a guarantee of integrity; if tampered with, the processor will erase its memory and cease to function. Thus, one can place customizable, trusted source code analyzers [10, **?**] in secure co-processors. Such a device can be installed as a co-processor at the site of the vendor; it can be sent an authenticated message which describes the type of analysis to be conducted. The smart-card resident tool can perform the analysis, and sign the results. The signature verifies that the results were created by trustworthy software resident in a secure machine, even at a potentially hostile site.

If physically protected, trusted tools are not available, and $\mathcal{V}$ is unwilling to disclose the source, it may become necessary to conduct test coverage verification using only the binary.

3.2 Verifying coverage with just the Binary

It possible to use just the binary to verify test coverage. In the verification protocol listed below, we use basic-block coverage as an illustration; this approach can be easily extended to branch coverage[5] as well. The approach used here depends upon analysis of the binary, which $\mathcal{T}$ has access to. Given a binary, and a knowledge of the instruction set of the architecture, one can readily determine the control flow structure of the binary, and find instrumentation points for basic block coverage and branch coverage. Without forcing $\mathcal{V}$ to reveal the source, $\mathcal{T}$ can generate a reliable coverage set.

Protocol 4

1. $\mathcal{V}$ derives coverage points $\{\ i, C_\gamma(i)\},\ for\ i = 1 \ldots \mid C_\gamma \mid$. using a coverage analysis tool on the binary; for each coverage point $C_\gamma(i)$ $\mathcal{V}$ identifies a test case t_i.

2. $\mathcal{V}$ sends to $\mathcal{T}$ the binary B_s, and $\{i, C_\gamma(i), t_i\},\ for\ i = 1 \ldots \mid C_\gamma \mid$.

3. $\mathcal{T}$ can use binary analysis to verify that the coverage set has been correctly derived by $\mathcal{V}$. Then, $\mathcal{T}$ can set a breakpoint at at each of the coverage points using binary instrumentation and execute tests to verify the coverage and correct operation.

[5] Extending this to more stringent criteria, such as dataflow coverage testing is beyond the scope of this paper.

The main challenge with the above Protocol is for $\mathcal{V}$ to find the test cases. However, $\mathcal{V}$ does have access to both the binary symbol table and the source code. Given a machine address, $\mathcal{V}$ can identify the corresponding source line easily, using a debugger (*e.g.,* with `gdb`, the "`info line` **addr*" will translate a given machine address to a source file and line number). If the $\mathcal{V}$ has previously developed a good covering test set, and verified his coverage levels, he can readily identify the specific covering test using the file/line number and data from his coverage verification process.

However, there are several difficulties with this approach; they all have to do with "testability" of the binary. Finding test cases to cover a given location in a binary can be harder than with source code, especially when the location occurs in either:

1. code generated by the compiler in response to a complex source language operator (e.g, inlined constructors or overloaded operators in C++); this code may contain control flow not present in the source, or in
2. code generated erroneously by the compiler, or in
3. COTS components incorporated by vendor in his product, for which the vendor has no tests;

There are approaches to dealing with some of these issues. Generated code often corresponds to idioms; this information can be used to find test cases. Sometimes generated code may contain additional control flow that represents different cases that can occur in the field, and $\mathcal{V}$ can legitimately be expected to supply covering test cases. When the generated code is genuinely unreachable [28], $\mathcal{V}$ can claim it as such, and supply source code that $\mathcal{C}$ can compile to create similar binaries. Occurrences of dead code in the binary are really bugs in the compiler, and are likely to be rare.

Even when a coverage point happens to fall within the bounds of an COTS binary, $\mathcal{V}$ has several options for test coverage verification. If the COTS is a well-known, reputable, piece of public domain software, $\mathcal{V}$ can simply identify the software, and $\mathcal{C}$ can download it and do a byte-comparison. Even if the COTS in not public, signatures can be obtained from V_{cots}, the vendor, for comparison. If the COTS is not well known, but has been independently subject to test coverage verification, then evidence of this verification can be provided to $\mathcal{C}$. Another approach is for $\mathcal{V}$ to relay the coverage points to V_{cots}, who may be able to handle them, and pass its responses back to $\mathcal{T}$.

Binary-based coverage verification can be performed without revealing source code or symbol tables, and without resorting to trusted tools. The trade-off here is that the mapping to source code may be non-trivial for some parts of the binary; for this and other reasons, it may be hard to construct an exercising test case. As binary de-compilation tools [8] mature and become more widely available, they can be used by customers to build confidence about areas of the binary that $\mathcal{V}$ claims to be non-testable for the reasons listed above.

Even so, $\mathcal{V}$ still has to reveal a lot to $\mathcal{T}$: the entire coverage set and the entire test set. If X is a very popular and/or difficult system to build, this information may be very valuable, and $\mathcal{T}$ may covertly sell this information to $\mathcal{V}$'s competitors. The next protocol reduces the amount of information $\mathcal{V}$ has to reveal, without increasing $\mathcal{V}$'s opportunities to cheat.

3.3 Verifying coverage with a sample of the test cases

We describe a sampling technique for using information about a small proportion of test coverage points to estimate the actual coverage on the entire program.

Protocol 5

1. $\mathcal{V}$ builds B_s from S, creates C_γ, $\sigma_{K_{\mathcal{V}}^{-1}}(i, C_\gamma(i), t_i, r_i)$, $for\ i = 1 \ldots \mid C_\gamma \mid$, t_i is the corresponding test case[6], and r_i is a random number inserted to blind the signature.
2. $\mathcal{V}$ sends $\mathcal{T}$ all the above signatures and B_s.
3. $\mathcal{T}$ challenges $\mathcal{V}$ with some small number l of the coverage points.
4. For each challenge, $\mathcal{V}$ reveals $C_\gamma(i), t_i$, and r_i. $\mathcal{T}$ can cross-check with the signatures delivered above.
5. $\mathcal{T}$ uses the coverage point location information, instruments B_s and runs the supplied test case to check coverage.
6. $\mathcal{T}$ signs the binary and sends $\sigma_{K_{\mathcal{T}}^{-1}}(B_s)$ to $\mathcal{V}$.
7. $\mathcal{T}$ archives the testing procedure including all the information sent in step 2.
8. $\mathcal{V}$ ships $(B_s, \sigma_{K_{\mathcal{T}}^{-1}}(B_s))$ to $\mathcal{C}$.

We shall discuss the commitments in step 1 presently; for now, we focus on steps 3-5. Here $\mathcal{T}$ randomly picks a small number of challenges to $\mathcal{V}$. This method betters all the above protocols in one important way: $\mathcal{V}$ reveals only *some* coverage points, and *some* tests. Since $\mathcal{V}$ cannot predict which coverage points $\mathcal{T}$ will pick, he must prepare tests to cover most of them. $\mathcal{V}$ packages the test cases with the corresponding coverage point, and a random number; this discourages $\mathcal{T}$ from brute-force searching for test cases and coverage points using the signatures from step 1; he reveals these on being challenged. With a small number of random challenges, $\mathcal{T}$ can bound $\mathcal{V}$'s test coverage. If $\mathcal{V}$'s responses cover a proportion p_s of $\mathcal{T}$'s challenges, $\mathcal{T}$ can estimate $\mathcal{V}$'s actual coverage p (using Hoeffding's version of Chernoff bounds for the "positive tail" of a binomial distribution, see [22], pp. 190-191):

[6] Or a string indicating the lack of a test case for this coverage point.

$$P(p_s - p \geq \epsilon) \leq e^{\frac{-n\epsilon^2}{2p(1-p)}} \ when\ p \geq 0.5 \tag{1}$$

For a 95% confidence level, we can bound ϵ:

$$e^{\frac{-n\epsilon^2}{2p(1-p)}} \leq 0.05$$

$$\epsilon = \sqrt{\frac{2ln(\frac{1}{0.05})p(1-p)}{n}}$$

Clearly, as n goes up, $\mathcal{T}$ gains confidence in his estimate p_s. Thus, at the 95% confidence level, $\mathcal{T}$ can reasonably conclude that an estimate of $p = 0.95$ is no more than 0.09 *too high* with about 25 samples. Experimental work [20, 27] indicates that branch coverage levels in the range of 80-90% have a high likelihood of exposing faults in the software. Estimated coverage levels in this range can give a customer high confidence that the $\mathcal{V}$'s testing has exposed a good number of faults. So it is in $\mathcal{V}$'s interest to allow $\mathcal{T}$ the largest possible number of challenges, with a very high expected p. Clearly, this will motivate $\mathcal{V}$ to achieve very high coverage levels. It is also important to keep in mind that this "random sampling" places limits on the accuracy of $\mathcal{T}$'s estimate of $\mathcal{V}$'s test coverage; essentially, we are trading off accuracy for the amount of information disclosed for $\mathcal{T}$. In cases where this trade-off is acceptable, this technique is applicable.

There is a threat to the validity of the confidence level calculation described above: the sampling process is not really a series of independent events. Executions of coverage points (blocks, branches, or functions) are often strongly correlated. Agrawal [5] shows how to determine the set of statically independent coverage points from the control flow graph by computing the post-dominator and pre-dominator trees of basic blocks. The leaves of such trees could be used to form an independent set of coverage points. However, there could still be dynamic dependencies between blocks which cannot be feasibly determined by static analysis. One way to quantify these effects is to use sub-samples and statistical tests to determine if the estimates are stable. This is can be complex, may require more samples, and can enlarge the confidence intervals. In a later protocol, we commit ahead of time to a coverage level; this avoids sampling difficulties.

Now we return to the commitments (signatures) in step 1. These reveal a signature on each element of the coverage set and the corresponding test case; $\mathcal{V}$ pads each signature computation with a random number to securely hide the association of the test case and the coverage point for the unchallenged coverage points.

3.4 Coverage verification without a trusted third party

We now turn to the problem outlined in **P.3** earlier. All the approaches described above rely upon a trusted third party ($\mathcal{T}$). However, it may sometimes be undesirable to use $\mathcal{T}$, for reasons of economy, secrecy, or delays.

A naive approach to coverage verification when a $\mathcal{T}$ is unavailable would be for $\mathcal{V}$ to rerun the verification protocols described above with each software buyer. This is undesirable. First, repeating the protocol with each buyer is expensive and slow. Second, $\mathcal{V}$ would reveal information to different, potentially adversarial parties who might collude to reverse engineer secrets about B_s. Third, since a potentially adversarial buyer is involved, there is a risk that the challenge points might be deliberately chosen to expose the most valuable information about $\mathcal{V}$'s software (described above as attack **S.2**, Section 2.4.8). For example, if $\mathcal{T}$ was forced to reveal (on a challenge from a customer c_1) some test cases that pertained to handling some unusual or difficult case in the input domain, other customers might collude with c_1 to probe other points in the same area of the binary to expose $\mathcal{V}$'s implementation/design strategies for dealing with some difficult cases.

Re-examining *Protocols* $1 \ldots 5$ listed in the previous section, it becomes clear that the main role played by $\mathcal{T}$ is to choose the challenging coverage points. We eliminate this role using autonomous pseudo-random sampling.

Protocol 6

1. $\mathcal{V}$ prepares the binary B_s, and $\sigma_{K_{\mathcal{V}}^{-1}}(i, C_\gamma(i), t_i, r_i),\ for\ i = 1 \ldots \mid C_\gamma \mid$.
2. $\mathcal{V}$ computes a well-known, published pseudo random function of a high entropy but public seed B_s to yield a *location control string*, $\mathcal{L}$. Successive byte groups of $\mathcal{L}$ are used to derive locations $l^1, \ldots, l^j$.
3. For each l_i, $\mathcal{V}$ reveals the test cases, random numbers and coverage points; call each revelation $\mathcal{R}_i$.
4. After some set of challenges l_i, $\mathcal{V}$ stops, and packages the $\mathcal{R}_i$'s and the $\sigma_{K_{\mathcal{V}}^{-1}}(i, C_\gamma(i), t_i, r_i)$'s, along with his release of B_s
5. $\mathcal{C}$ verifies test coverage by repeating the generation of the location control string, and checking the corresponding revelations by $\mathcal{V}$ for coverage.

Protocol 6 offers several advantages. We have eliminated the "middleman," $\mathcal{T}$, thus saving time and money. This approach is also advantageous where secrecy is involved. Instead of $\mathcal{T}$, a one-way hash function now drives the choice of the challenges. $\mathcal{V}$ cannot control the value of the string S_{vc}. A customer can easily verify the value of the location control string using the delivered software and the public hash function. Furthermore, there is no need for $\mathcal{V}$ to repeat this process with each customer. There is no risk that customers might collude and pool information.

3.5 Residual Disclosure Concerns

We now return to the issue **P.2** raised in Section 2.4.5, How much of $\mathcal{V}$'s valuable technical secrets do our techniques reveal?

At a minimum, $\mathcal{V}$ has to ship the binary B_s. Simply from the binary (even without a symbol table) an adversarial customer $\mathcal{C}_\mathcal{A}$ can construct a good deal of information by static analysis: the control flow graph, the size of the data space, the number of functions/entry points, etc. A limited amount of static control and data dependency analysis is even possible. Indeed, tools like EEL [25] can perform much of this analysis. In addition, by instrumentation, and dynamic analysis, $\mathcal{C}_\mathcal{A}$ can detect which paths of the control flow graph are activated for different input conditions. Work by Ball & Larus [7] shows how it is possible to trace and profile control flow path execution using just the binary. Additional information can be gained by tracing memory references and building dynamic slices. Given the degree of information that can be reconstructed, it is important to evaluate carefully whether the approaches listed above yield additional opportunities for $\mathcal{C}_\mathcal{A}$.

First, assume that the symbol table is stripped from the delivered binary B_s. During the verification process, whether driven by $\mathcal{T}$ or not, the only additional information revealed in response to challenges are the relevant coverage points and the applicable test cases. The coverage points, if based on the binary, can be independently generated by $\mathcal{C}_\mathcal{A}$ or $\mathcal{T}$. The only additional information is the test case, and its connection to this coverage point. The value of this information can vary. Important factors include the manner in which the test cases are supplied, the resulting behavior of the system, etc. Test cases could be supplied in the form of source code or ASCII input (which might be very revealing) or in the form of binary objects or binary data (which could be more difficult to interpret). As far as the verification protocol is concerned, the only relevant behavior is that the selected challenge coverage point be exercised. However, there may be additional behavior that reveals information to the adversary.

In the best case, if the relevant test case reflects a fairly "typical" type of usage, then the information given away is minimal. Presumably $\mathcal{C}_\mathcal{A}$ would very likely find this out during his attacks using dynamic analysis. However, if the test case reflects an unusual circumstance, and the test case is delivered in a manner transparent to $\mathcal{C}_\mathcal{A}$, then some valuable information about unusual but important design and requirements details of B_s may be revealed. The risk of such an exposure is traded off against the ability to verify test coverage.

The delivery of test cases to $\mathcal{C}_\mathcal{A}$, particularly the tests that somehow embody valuable proprietary information is an important issue that remains to be addressed. Can we deliver test cases in a way that protect $\mathcal{V}$'s secrets, while still exhibiting test coverage?

Now we relax the assumption that the symbol table is stripped out. While it is possible to verify test coverage without the symbol table, there are some difficulties associated with omitting it. In several of the protocols we listed earlier, we assumed that the symbol table was included in the binary shipped to the verifier. Clearly, the symbol table offers additional opportunities for $\mathcal{C}_\mathcal{A}$ to reconstruct information. Some standard obfuscation techniques such as garbling the symbolic names would be helpful. In general, however, the advantages of omitting the symbol table may override the resulting difficulties.

Summary In this section, we described various approaches to test coverage verification

which both reduced the disclosure required by the vendor $\mathcal{V}$ and used cryptographic techniques to reduce $\mathcal{V}$'s ability to cheat. Specifically, signature-based commitments made it impractical for $\mathcal{V}$ to try bait & switch attacks (*P.1.a,b*). Trusted tools and binary analysis make it difficult for the vendor to ship incorrect coverage sets (*P.1.b*). Interactive selection (sampling) of a subset of the coverage points (a sample) by $\mathcal{T}$, in addition to the above techniques, reduce the amount of disclosure by $\mathcal{V}$ (**P.2**) Finally, autonomous pseudo-random sampling reduces or eliminates the role of $\mathcal{T}$, thus reducing $\mathcal{V}$'s costs (**P.3**). These last two techniques rely on statistical estimates of coverage levels based on the samples, and are vulnerable to attacks that exploit the confidence intervals associated with the statistical estimates. We now describe some techniques to thwart such attacks.

4 Defending against Sampling Attacks

In this section, we discuss the sampling attacks **S.1** and **S.2** first described in Sections 2.4.7 and 2.4.8. In the last approach described above, the vendor generates a location control string in a verifiable manner, and uses that to drive the sampling process. There is a risk in this scenario. $\mathcal{V}$ has control over the input to the function that generates the control string. Therefore the $\mathcal{V}$ could automatically repeat the following:

1. Compute $\mathcal{L} = hash(B_s)$.
2. If a very large subset of the resulting locations $l_1 \ldots l_n$ are not covered, stop. Otherwise,
3. Generate another binary B_{s1} by padding B_s with null instructions. Go to step 1

This amounts to repeated Bernoulli trials drawn from the set of coverage points. After some trials, $\mathcal{V}$ could find an $\mathcal{L}$ that artificially boosts $\mathcal{V}$'s coverage ratio. To avoid this attack, we need a way to "monitor" such trials. Thus, we force the vendor to archive all the trials with a monitor. This lets us bound the expansion of the confidence if $\mathcal{V}$ makes several attempts at verification.

4.1 Monitoring verification trials with a trusted archive

We introduce a trusted archive, $\mathcal{T}_\mathcal{A}$, to record the fact that $\mathcal{V}$ initiates testing. We assume $\mathcal{T}_\mathcal{A}$ archives *all* information submitted to it, and can provide signed responses to queries concerning archive contents and history.

Protocol 7

1. When ready to verify coverage, $\mathcal{V}$ "registers" the following with $\mathcal{T}$: 1) a query of historical usage of the archive, 2) identifying string I of the program to be tested, 3) $\sigma_{K_\mathcal{V}^{-1}}(B_s, I)$, and 4) $\sigma_{K_\mathcal{V}^{-1}}(i, C_\gamma(i), t_i, r_i),\ for\ i = 1 \ldots \mid C_\gamma \mid$.

2. $\mathcal{T}_{\mathcal{A}}$ acknowledges $\mathcal{V}$'s registration with the registration history and a signature of the history. For simplicity, we assume that the returned signature is random for each query response even if the query is the same. For example, a parameter of the signature function may include a random number.

3. $\mathcal{V}$ follows a procedure like Protocol 6 except that: in step 2) $\mathcal{V}$ computes the location control string by seeding a well-known, published pseudo-random number generator using the signature returned by $\mathcal{T}_{\mathcal{A}}$. In step 4), $\mathcal{V}$ also packages the identifying string, and the historical usage of $\mathcal{V}$ signed by $\mathcal{T}_{\mathcal{A}}$. In step 5, $\mathcal{C}$ uses the historical information from the archive to determine the $\mathcal{V}$'s use of the archive. Also, $\mathcal{C}$ verifies $\mathcal{T}_{\mathcal{A}}$'s signature on the testing profile and the signature that seeded the pseudo-random process.

Our approach involves a string identifying the system that is also archived. When releasing a trace of the test coverage verification protocol, this identifying string is included in the release, and is input to the process that generates the location control string. $\mathcal{V}$ is free to generate many location control strings, but each time, step 1 above requires him to register with $\mathcal{T}_{\mathcal{A}}$. Each registration become a part of his history that is available to customers. Given a release from $\mathcal{V}$ that incorporates a system (with identifying string), and a verification trace, $\mathcal{C}$ can query $\mathcal{T}_{\mathcal{A}}$ to find registrations from $\mathcal{V}$. Let us assume that there are m such registrations with similar identification strings. Further assume that the location control string checks out, and that $\mathcal{V}$ presents the most favorable value of p_s and includes n challenges, of which a proportion p are found to be covered by running the test cases provided by $\mathcal{V}$. Given this data, $\mathcal{C}$ can estimate the the probability that p differs from actual coverage level p_a by ϵ, using m disjoint occurrences in the inequality 1:

$$P(p_s - p \geq \epsilon) \leq m e^{\frac{-n\epsilon^2}{2p(1-p)}}$$

Therefore, for a 95 % confidence level, we can estimate the error ϵ as follows:

$$\epsilon = \sqrt{\frac{2ln(\frac{m}{0.05})p(1-p)}{n}}$$

Recall from inequality 1: that with about 25 samples, one can achieve a 95% confidence level with an error ceiling of 0.09 on an estimate of $p = 0.95$. With 5 registrations, this error ceiling increases to 0.12. With more registrations, the $\mathcal{C}$'s faith in the actual coverage results from a particular verification trace is limited. It can seen that it is in the interests of $\mathcal{V}$ to a) delay registration for test coverage verification until enough tests have been developed for a high coverage level, b) provide clear, distinguished identifying strings for each of his software offerings so that customers don't conflate registrations for different products and c) limit the number of registrations for any particular piece of software[7]. For a vendor whose development processes already achieve high levels of coverage, this not a significant burden. Finally, $\mathcal{T}_{\mathcal{A}}$'s services are quite limited, fully automatable, and thus should be inexpensive.

[7] Unless, of course, he reveals the verification trace for each of them, in which case the customers can get very tightly bounded estimates of his coverage level.

4.2 Committing and Enforcing an Upper Bound on Test Coverage

The protocols described up to this point have a weakness: $\mathcal{V}$ may get lucky and demonstrate a higher test coverage then he actually has. This is inherent to challenges based on random sampling. We now describe a technique that requires the vendor to assert an upper bound on test coverage. This technique can be used in conjunction with any of the protocols described above. With this approach, the vendor can be caught cheating if he is called upon to reveal a test case corresponding to a coverage point which he did not account for as a untested coverage point. However, the vendor is not forced to reveal potentially sensitive information about exactly which coverage points have no test cases. We present the technique here as a series of steps that can be interleaved into the protocols described above.

1. $\mathcal{V}$ commits to untested coverage points by sending $Hash(i, r_i, C_i), i = 1 \dots N_{nt}$ for each coverage point not tested (where r_i is chosen at random by $\mathcal{V}$). Using this, the $\mathcal{C}$ or $\mathcal{T}$ can compute the upper bound on test coverage claims.
2. When the vendor is called upon to reveal a test point for which it does not have a test case, the vendor reveals r_i and i, the reference to the particular hash in the first step. The tester can recompute the hash of the tuple $i, r_i,$ and C_i and compare it to the commitment of the untested coverage points.
3. If testing results with numbers higher than the coverage claims, the results are decreased to the asserted upper bound.

In step 1, $\mathcal{V}$ commits to all the coverage points which are admittedly not covered by test cases. From this information $\mathcal{C}$ (or $\mathcal{T}$, as the case might be) can determine an upper bound on the actual coverage ratio. For example, in the case of Protocols 6 and 7, $\mathcal{C}$ can determine the coverage set by analysis of the binary. If the size of this set is N_{cs}, $\mathcal{C}$ can bound the coverage ratio is:

$$\frac{N_{cs} - N_{nt}}{N_{cs}}$$

Step 1 can be done at the same time as the first step in Protocols 6 and 7. Step 2 above is basically an extension to the random challenge step in many of the protocols. Given a random challenge, $\mathcal{V}$ may or may not have a test case; if $\mathcal{V}$ does, the protocols work as described earlier. In the case where there is no test case, $\mathcal{V}$ reveals r_i and i, thus "checking off" one of the uncovered coverage points committed in step 1. If $\mathcal{V}$ is unable to reveal a test case, or an r_i, i pair, $\mathcal{V}$ is caught in a lie. Finally, in Step 3 above, the $\mathcal{C}$ can compare the estimate from the random sample to the *a priori* upper bound computed above.

In this technique, $\mathcal{V}$ makes a clear claim about the proportion of tests covered, and the total number of tests. The purpose of the random trials therefore is not to narrow the confidence intervals around an *estimate* the value of the coverage ratio, but just to make sure $\mathcal{V}$ is not lying. With each trial, there is a probability that $\mathcal{V}$ will be caught; this increases with the

number of trials. To model this analytically, assume that there are N total coverage points, and $\mathcal{V}$ is lying about l of those. i.e., for l of those $\mathcal{V}$ has no covering tests, but is pretending that he does. Denote the fraction $\frac{l}{N}$ by f. On any one trial, the chance that $\mathcal{V}$ will be caught is f, and that $\mathcal{V}$ will sneak through is $1 - f$. After n trials, the probability that $\mathcal{V}$ will escape[8] is

$$(1 - f)^n$$

Let's bound this above by ϵ:

$$(1 - f)^n \leq \epsilon$$

$$i.e. \;\; n \, log \, (1 - f) \leq log \, \epsilon$$

$$i.e., \;\; n \; \leq \frac{log \, \epsilon}{log \, (1 - f)}$$

With $f = 10\%$, (i.e., $\mathcal{V}$ is lying about one-tenth of the coverage points), there is a 95% chance (i.e., $\epsilon = 0.05$) that $\mathcal{V}$ will be caught after roughly 28 random challenges. Again, it behooves $\mathcal{V}$ to provide many trials to build confidence that $\mathcal{V}$ is not lying. Note that the value of the coverage ratio is always what the vendor says it is—the confidence value refers to $\mathcal{C}$'s subjective probability of $\mathcal{V}$'s veracity, i.e., the likelihood that $\mathcal{V}$ would have been caught trying to cheat. Such an event will seriously damage a vendor's credibility; most vendors may not be willing to tolerate even a small chance of being caught, and thus would be cautious about misrepresenting the coverage ratio. If $\mathcal{V}$ tried to cheat on even 5% of the cases, with 20 trials, there is a 50% chance of exposure. For many vendors, this may be intolerable. We expect that this approach should provide $\mathcal{C}$ with a more accurate coverage estimate.

It should be noted that this approach is susceptible to prospecting attacks— the vendor conducting trials repeatedly to "get lucky" and slip by with false claim of high coverage. This threat can be reduced by combining coverage ratio commitment with use of a trusted archive described in Protocol 7.

5 Defending against testing-related attacks

This section describes methods to counter the testing related attacks **T.1** and **T.2**. It should be noted here that while coverage testing is widely used in industry, some researchers dispute the effectiveness of white-box (or "clearbox") coverage methods. Most recent empirical work [20, 27] has found that test sets with coverage levels in the range of of 80-90% have a high chance of exposing failures. Earlier work [15] had yielded inconclusive results; however, the programs used in [15] were substantially smaller than [20, 27]. On the analytic front,

[8]For simplicity, we assume trials can be repeated.

rigorous probabilistic models of the relationship between increasing white-box coverage and the likelihood of fault detection have been developed [17]. However, no known testing process is perfect. All known methods, whether white-box or black-box, *will let some faults slip!* The best current experimental work [20, 27, 14] suggests that high levels of white-box test coverage can guarantee high levels of fault detection. However, since white-box testing is not perfect, there are several complications. Given a coverage point that has faults, there may be several test cases that exercise that point. Some of these test cases will expose faults, but others may not. Consider a particular coverage point c, which has a fault f. When $\mathcal{V}$ generates a covering set of test cases, assume a test τ is found which just happens to not expose the fault f. Since the software tests correctly, $\mathcal{V}$ will simply supply this test case along with the (faulty) delivered program. No malice is intended; in this particular case, coverage testing is imperfect, and the vendor honestly delivered a program with a fault that just happened to slip by.

We will now discuss attack **T.1**. Consider the case where the test case chosen by $\mathcal{V}$ happens to expose the fault. $\mathcal{V}$ now has two choices: fix the fault, or cheat. $\mathcal{V}$ can try to find a different test case τ^*, which covers c but does not expose a fault. The incentive for $\mathcal{V}$ to cheat depends on several factors: the likelihood the fault will occur in the field, the market conditions for the software (how many copies can be sold, for how long?), the cost of fixing the software, and the difficulty of finding a fault-hiding (but c-covering) test case τ^*. In most cases, it will probably be best to fix the fault. In the absolute worst case, assuming that the nature of the fault and the business conditions really motivate $\mathcal{V}$ to cheat, the cost finding such a τ^* depends on the distribution of failure-causing input within the subdomain of the input that causes the coverage point c to be exercised. If $\mathcal{V}$ is lucky, the failure region is small and well isolated within this input partition; he may succeed in finding a covering test case that is fault-hiding. On the other hand, if the fault is provoked by many inputs from the subdomain of the input that exercises c, then $\mathcal{V}$ may have to read the code carefully (or spend a lot of time randomly sampling the input partition) to find such a τ^*; in this case, it may be easier to simply fix the problem. Finally, this method of attack (finding covering test cases that hide the fault) is not specific to the cryptographic techniques we have described; even the third party coverage certification method that is currently used is vulnerable.

We can favorably bias the situation by encouraging vendors to provide (and commit to) more than one test case per coverage point. The more tests a vendor provides for a coverage point, the less likely it is that *all* these tests are fault-hiding. A random search for fault-covering cases in the corresponding input partition is not likely to work; it becomes necessary to understand the nature of the fault, and carefully construct several examples that exercise that coverage point, but hide the fault. As this effort increases, so does the incentive for $\mathcal{V}$ to simply fix the fault. The more test cases $\mathcal{V}$ can provide[9] for each coverage point, the less likely it is that a fault is deliberately hidden in that coverage point.

Another difficulty inherent in coverage analysis is the opportunity to *pad* (attack **T.2.**).

[9]The test cases do not all have to be revealed; they could be hidden as a hash value, and revealed only upon a (random) challenge.

Vendors can add spurious code which introduces additional control flow. Such code can be designed to be readily covered, thus artificially boosting coverage. It is not feasible to determine if this has been done. The only way to totally avoid this problem is to insist upon 100% coverage for some criteria (padding becomes irrelevant). For some criteria 100% coverage may be feasible for a medium-sized component; since such coverage levels are indicators of thorough testing, there may be market incentives that push vendors to achieve such a level. A series of coverage levels for increasingly stronger criteria, starting at and gradually decreasing from 100%, would be a desirable goal. Another approach to discourage padding is to demand explanations when challenge points are not covered by tests. A vendor can voluntarily build confidence that he hasn't padded by making large number of pseudo-random choices among his uncovered set and providing explanations for why they are not covered, and describing the conditions under which the points would be executed. Such conditions had better be highly unusual and difficult to create; if they are not, the vendor could be expected to provide a test case. If a large number of such disclosures are made, the vendor would be at risk of embarrassment by subsequent revelation that the coverage point was executed under less uncommon circumstances. A large number of such disclosures can build confidence that points do not remain uncovered simply as a result of padding. Again, another powerful disincentive to excessive padding is the vendor's inherent desire to produce high-quality software, and thus avoid costs of refunds, cancelled sales, increased technical support, and extra update releases.

To summarize, our work rests on the assumption (supported by [20, 27, 14]) that comprehensive coverage testing tends to expose faults, and on the assumption that vendors will most often find it more profitable to fix faults exposed by a covering test (rather than searching for a test that covers but hides faults). In the worst case, when the difficulty of fixing the faults exceeds the difficulty of finding test cases to hide the fault, and $\mathcal{V}$ expects the faults in question are rare enough so that he can collect enough revenue before the fault is exposed in in the field, then he may be tempted to find a hiding test case. By encouraging vendors to reveal many test cases for each coverage point, we can decrease the incentive to hide faults. But even in this worst case, the use of these techniques can provide customers with the justified belief that the vendors have every incentive and the means to find and fix all but the ones that are very unusual and/or difficult to fix. Padding is another problem; 100% coverage is the best way to preclude the chance of padding. We have suggested some ways that vendors can provide evidence that they did not pad. We are actively pursuing better ways of determining if padding has occurred.

In any case, during practical use of these coverage verification protocols, it is important for both customers and vendors to be mindful of the suspected limitations of white-box coverage testing. Additional black-box and/or acceptance testing will often be needed.

6 Tool Support

We have constructed some tools for use in conjunction with the protocols described above. These tools are publicly available. These tools are intended to be used by a vendor who has already done the work of creating a comprehensive set of test cases, and wishes to use the protocols described above to provide evidence of test coverage.

Source Code Based Tools As described above in Protocols 2 and 3, coverage points can be determined by analysis of the source; the actual instrumentation can be done on the binary. To support this, we have created a tool, `src2binmod` , which analyzes the source code and creates a set of commands to place instrumentations. This tool is built with the `gen++` incarnation of GENOA [10] and is compatible with C++ and ANSI C source files. It creates output files that can be used to conduct instrumentation with ATOM [34] or with `gdb`. In addition, it also creates a command file for `gdb` that can be used to generate actual binary positions; this can be used to perform instrumentations on stripped binaries without symbol table information (Protocol 3). This tool can be obtained in source code from the first author; the `gen++` tool itself is also available for download [11].

Binary Based Tools We have also built a tool, `bin2chall` , for conducting sampling-based coverage testing (Protocols 5 and others) directly on the binary using pseudo-random sampling. This tool works by first running the UNIX tool `size` to determine the bounds of the executable (text) portion of the binary. It then seeds a random number generator using a supplied seed, and generates a series of challenges in the form of `info line *0xXXXX` commands to `gdb`, where `0xXXXX` is a hexademical address within the bounds defined by the output of the `size` command. The vendor $\mathcal{V}$ can load a binary (with symbol table) into `gdb` and use these commands to generate source line information; with this information, and the tests he already has, s/he can find the tests to cover each of the addresses. `bin2chall` also generates commands `break *0xXXXX` corresponding to each of the `info` commands above; these commands are for use by $\mathcal{T}$ or $\mathcal{C}$ to instrument a (stripped) binary. By using a publicly known, secure, pseudo-random number generator, and by sharing the seed, the vendor can guarantee that the sampling was done fairly. This tool can be downloaded in source form [1]; it can be modified for use in conjunction with any cryptographically strong pseudo-random generator.

7 Conclusion

We have shown a set of protocols that can be used to verify test coverage, while protecting information valuable to the vendor and simultaneously reducing the vendor's ability to cheat. These protocols use various techniques such as trusted third parties, trusted tools, signatures, random challenges, and "autonomous" random challenges. These techniques can be used in different combinations, depending on the needs of the customer and the vendor. Combinations other than the ones we have presented are possible. For example, the tech-

niques of Protocol 6 and 7 can be used in combination to precisely bound coverage level, and use a trusted archive to prevent prospecting attacks by $\mathcal{V}$.

Some of our techniques are compatible with a widely used method for repelling reverse engineering, which is shipping binaries without source code or a symbol table. The only additional vendor information that our techniques need reveal are a small proportion of test cases. While it is possible to conceive of reverse engineering countermeasures that may not be compatible with our techniques, we believe that we can adjust our methods to be compatible with countermeasures that may complicate reverse engineering, such as the introduction of additional static control flow, additional dynamic control flow, or even certain other approaches that involve dynamically decrypting code prior to execution; we are actively exploring these issues.

Finally, we note that "self-validated" approaches (if they can be perfected) that verify testing effectiveness may have an important *leveling* effect on the software market: small, relatively unknown software vendors with limited resources can provide credible evidence of high-quality processes, and thus compete with much larger corporations.

Notice that our protocols impose a modest overhead on developers who already have achieved high levels of test coverage: with the self-validated approach of Protocol 6, the developer simply runs an autonomous random sampling of his coverage points; his (pre-existing) test coverage data points to the relevant tests to be revealed. Much of this can be automated, and we have constructed supporting source-based tools using GENOA [10], and a tool for binary based sampling.

Using such methods, any vendor, without the help of a trusted third party, and at relatively low overheads, can provide a credible claim that their software quality control is stringent, while disclosing only a minimal amount of information. This enables small and unknown vendors to compete effectively (on the basis of perceived quality) with very large vendors with established brand names. It is our hope that such approaches, as they are perfected and widely adopted, will engender a creative "churn" in the software market place, to the ultimate benefit of the consumer. J. Gannon [19] has pointed out a significant potential application of this work: customers with stringent quality requirements, such as the nuclear industry [31] need to obtain good evidence of good software quality control practices, prior to "dedicating" COTS software for use in safety-critical control systems. Our techniques can be used to obtain such evidence while protecting certain trade secrets. Well-tested components can certainly provide an additional level of confidence for designers of safety-critical systems. However, testing alone is often not sufficient quality assurance for safety critical systems. Other forms of formal verification may be necessary. Finally, even with the use of COTS components for which high levels of test coverage can be verified, it may be necessary to use additional techniques such as "wrapping" [13, 36] to ensure high levels of safety and reliability.

Acknowledgements We are grateful to Naser Barghouti, Alex Borgida, Mary Fernandez, Richard Kemmerer, Douglas McIlroy, Dewayne Perry, Robert Schapire, Elayne Weyuker,

Pamela Zave, and the anonymous reviewers of ACM SIGSOFT *FSE '97* and IEEE TSE for valuable comments and technical input.

References

[1] Source code for `bin2chall`. `http:/www.cs.ucdavis.edu/~devanbu/bin2chall.-cpio.gz`.

[2] *The Mondex Magazine, Mondex International Limited, July 1997.* (See also: `http://www.mondex.com`).

[3] Guideline for automatic data processing and risk analysis FIPS 65. National Bureau of Standards, Gaithersburg, MD, USA, August 1979.

[4] Delta Software Testing (accredited by Danish Accreditation Authority-DANAK). `http://www.delta.dk/se/ats.htm`.

[5] H. Agrawal. Dominators, super blocks and program coverage. In *Proceedings, POPL 94*, 1986.

[6] R. Atkinson. Security architecture for the internet protocol. request forcomments (proposed standard) rfc 1825, August 1995.

[7] T. Ball and J. Larus. Efficient path profiling. In *Micro '96*. IEEE Press, December 1996.

[8] C. Cifuentes. Partial automation of an integrated reverse engineering environment for binary code. In *Third Working Conference on Reverse Engineering*, 1996.

[9] C. Cifuentes and J. Gough. Decompilation of binary programs. *Software Practice and Experience*, July 1995.

[10] P. Devanbu. Genoa—a language and front-end independent source code analyzer generator. In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992.

[11] P. Devanbu. The GEN++ page. `http://seclab.cs.ucdavis.edu/~devanbu/genp`, 1998.

[12] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4), July 1984.

[13] Fred B. Schneider (EDITOR). *Trust in Cyberspace*. National Academy Press (Computer Science and Telecommunications Board, National Research Council), 1998.

[14] P.G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, November 1998.

[15] P.G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, August 1993.

[16] P.G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, August 1988.

[17] P.G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, March 1993.

[18] A. Freier, P. Karlton, and P. Kocher. The SSL protocol, version 3.0 (internet draft), March 1996.

[19] John Gannon. Personal conversation, April 1997.

[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering*. IEEE Computer Society, May 1994.

[21] Plum Hall Inc. `http:/www.plumhall.com`.

[22] M. J. Kearns and U. V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994.

[23] National Software Testing Labs. `http://www.nstl.com`.

[24] Software Testing Labs. `http://www.stlabs.com`.

[25] J. Larus and E. Schnarr. Eel: Machine-independent executable editing. In *ACM SIGPLAN PLDI*. ACM Press, 1995.

[26] J. Lidiard. Can COTS software be trusted? *Interface: Newsletter of the FAA Software Engineering Process Group*, November 1996.

[27] Y. Malaiya, N. Li, J. Bieman, R. Karcich, and B. Skibbe. Software test coverage and reliability. Technical report, Colorado State University, 1996.

[28] Doug McIlroy. Personal e-mail communication, 1996.

[29] Alfred J. Menezes, Paul C. van Oorschot, Scott, and A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[30] P. Newmann. *Computer Related Risks*. Addison Wesley, 1995.

[31] Committee on Application of Digital Instrumentation, Control Systems to Nuclear Power Plant Operations, and Safety. *Digital Instrumentation and Control Systems in Nuclear Power Plants—Safety and Reliability Issues–Final Report*. National Academy Press (Board on Energy and Environmental Systems, National Research Council), 1997.

[32] N. Ramsey and M. Fernandez. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 1997.

[33] J. A. Scott, G. G. Preckshot, and J.M. Gallagher. Using commercial off-the-shelf (COTS) software in high-consequence safety systems. Technical Report UCRL-JC-122246, Lawrence Livermore National Laboratory, 1995.

[34] A. Srivastava and A. Eustace. Atom: A tool for building customized program analysis tools. Technical Report 1994/2, DEC Western Research Labs, 1994.

[35] Applied Testing and Technology Inc. `http://www.aptest.com`.

[36] J. M. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6), 1998.

[37] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[38] Bennet Yee and Doug Tygar. Secure coprocessors in electronic commerce applications. In *Proceedings of The First USENIX Workshop on Electronic Commerce*, New York, New York, July 1995.